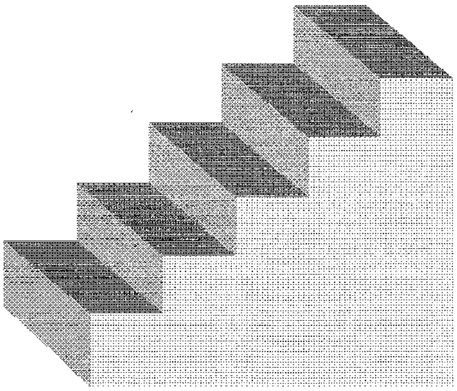

PROGRAMMING IN FORTRAN

Fourth Edition

Ömer Akgiray

e-mail:

omer.akgiray@marmara.edu.tr

CHAPTER 2: *DECISION STRUCTURES*

2.1 Logical Constants and Variables

There are only two **logical constants**. They are *true* and *false*, and are represented in FORTRAN as `.TRUE.` and `.FALSE.`, respectively. Note that the words `TRUE` and `FALSE` are preceded and followed by periods. When logical data are printed, the letters `T` and `F`, representing `.TRUE.` and `.FALSE.`, respectively, are printed without the enclosing periods. Storage locations which are used to store logical values which can be varied are called **logical variables**. Logical variable names must be declared by the type statement `LOGICAL`. For example,

```
LOGICAL  A, MOON, GOOD
```

declares `A`, `MOON`, and `GOOD` as logical variables. Examples of the use of logical constants and variables will be given later in this chapter.

2.2 Relational Expressions and the Single-Alternative Decision Structure

FORTRAN provides a decision-making capability in the form of a construct known as the **block-`IF` structure**. The simplest form of this structure is the **single-alternative decision form** and has the following general format:

```
IF(logical expression)THEN
    Task
ENDIF
```

A *logical expression* (sometimes also called a *logical condition*) is an expression which is either true or false (i.e., `.TRUE.` or `.FALSE.`). When the block-`IF` structure is executed, the *logical expression* is first evaluated. If the expression is true, then the program statements that constitute *Task* are executed. If the *logical expression* is false, then *Task* is skipped and execution continues with the first statement following the `ENDIF` statement.

We shall first discuss a special class of logical expressions, namely **relational expressions**. A relational expression consists of two arithmetic expressions connected by a single **relational operator**. The following are the relational operators used in FORTRAN:

Relational Operator	Description
`.EQ.`	equal to
`.NE.`	not equal to
`.GT.`	greater than
`.LT.`	less than
`.LE.`	less than or equal to
`.GE.`	greater than or equal to

It should be noted that each relational operator has four characters, two letters preceded and followed by a period[1]. Some examples of mathematical conditions and their equivalent FORTRAN relational expressions are given below:

$A < B$	`A.LT.B`
$A^2 \neq B$	`(A**2).NE.B`
$(A + 2B) \geq \sqrt{C}$	`(A+2*B).GE.SQRT(C)`

Assume `A`, `B`, and `C` have been assigned the values `1.0`, `2.0`, and `100.0`, respectively. Then, the first two relational expressions above have the value `.TRUE.`, the third has the value `.FALSE.`.

Before studying example programs employing the `IF` statement, it will be appropriate here to introduce the **`STOP` statement**. A `STOP` statement can be used to terminate program execution before reaching the `END` statement. This statement is often used when a program has detected some error from which it cannot recover. It has the following general form:

`STOP` *message*

where the optional parameter *message* is either a character constant or an integer constant between 0 and 99999. For example, the statement

```
      STOP 'Error detected!'
```

displays the message `Error detected!` and terminates programs execution.

Note that the `STOP` statement and the `END` statement are not equivalent. As noted before, every FORTRAN program unit (including subprograms) must contain exactly one `END` statement. The `STOP` statement, on the other hand, may be used at more than one point within a program or not used at all.

[1] When FORTRAN was being developed in 1950s, it was not possible to punch signs such as < and > onto cards. As a result, all relational operators consisted of two letters enclosed between periods.

Before FORTRAN 77, it was necessary to use a STOP statement somewhere in the program (not necessarily at the end) to terminate execution. The END statement was not an executable statement and its only function was to signal to the compiler that there were no more lines to be compiled[2]. When the execution sequence of the program statements is rather straightforward, it often happens that execution is terminated at only one place, i.e. at the end of the program. In such a case, before the advent of FORTRAN 77, one would have to place a STOP statement just before the END statement. Although many FORTRAN programmers still have the habit of placing a STOP statement just before the END statement of the main program, the STOP statement is not necessary if it is immediately followed by the END statement.

Note also that the END statement of a subprogram has the same effect as a RETURN statement, whereas the END statement in the main program has the same effect as a STOP statement.

Example 2.1:

Suppose we want to write a program that reads a number interactively, calculates the square root of the number, and displays the result on the computer screen. The intrinsic function SQRT will be used for this purpose. The square root of a negative real number, however, is not defined. It would therefore be useful to detect a negative number and print an informative error message when a negative number is entered by the user of our program.

```
      PROGRAM SQROOT
C***********************************************************************
C   Program to calculate the square root of a real number
C   NUM: A real value entered by the user
C***********************************************************************
      IMPLICIT NONE
      REAL NUM
      PRINT*, 'Enter your number: '
      READ*, NUM
      IF(NUM.LT.0) THEN
          PRINT*, 'Square root of a negative number is not defined.'
          STOP
      ENDIF
      PRINT*, 'Square root is', SQRT(NUM)
      END
```

Two sample runs of the program are given below:

```
Enter your number: 4
Square root is    2.00000

Enter your number: -1
Square root of a negative number is not defined.
```

Fortran 90 allows the use of <, <=, >, >=, ==, and /= instead of .LT., .LE., .GT., .GE., .EQ., and .NE., respectively. Note that the older forms .LT., .LE. etc. are still valid in Fortran 90.

[2] Organick and Meissner, p.76.

The program first displays a prompting message, informing the user that a number is to be entered. Once the value of `NUM` is read, it is compared with zero. Note that the relational expression

```
NUM.LT.0
```

compares a real variable (`NUM`) with an integer constant (`0` without a decimal point). This is permissible because mixed-type expressions are allowed in FORTRAN 77[3]. The compiler handles such a mixed-type logical expression as follows. The integer constant is first converted to type real by appending a decimal point. Then comparison between two real values are carried out.

If `NUM` is zero or positive, then the relational expression `NUM.LT.0` has the value `.FALSE.`, and the program execution continues with the statement immediately following the `ENDIF` statement:

```
PRINT*, 'Square root is', SQRT(NUM)
```

The program execution is then terminated by the `END` statement. If, on the other hand, `NUM` is negative, then the value of the relational expression `NUM.LT.0` will be `.TRUE.`, and the statements within the block-`IF` structure are next executed. The first statement

```
PRINT*, 'Square root of a negative number is not defined.'
```

gives information to the user about why the square root of the specified number cannot be calculated. Next, the program execution is terminated by the `STOP` statement. Notice that, in this case, the `PRINT` statement that follows the `ENDIF` statement is never executed. This example illustrates how the `STOP` statement can be used to terminate the execution of a program before reaching the `END` statement.

Although in this text we shall always indent the statements that constitute the body (i.e., *Task*) of a block-`IF` structure, this is of course not required. It is, however, a good practice to follow as it enhances the readability of programs. This point will become more apparent as we start using more complicated control structures such as the multiple-alternative decision structure (Section 2.3), and especially nested block-`IF` statements (Section 2.6) and nested loops (Chapter 3).

2.3 The Multiple-Alternative Decision Structure

The **double-alternative decision structure** has the following general format:

```
IF (logical expression) THEN
     Task
ELSE
     False Task
ENDIF
```

The *logical expression* is first evaluated. If the expression is true, then the program statements that constitute *Task* are executed and those that constitute *False Task* are skipped. If the *logical expression* is false, then *Task* is skipped and *False Task* is

[3] Expressions of the mixed-type were not permitted in standard FORTRAN 66. Be that as it may, most of the compilers commercially available in early 1970s allowed mixed-type expressions (Organick and Meissner, p.59 and p.276).

executed. In either case, execution continues with the first statement following the ENDIF statement.

Example 2.2:

In this example we look at a subprogram that determines if a given integer is even or odd. Since there are only two possible answers (even and odd), it seems reasonable to implement this subprogram as a logical function which can return two possible values (.TRUE. or .FALSE.). Consider the following function which returns the logical value .TRUE. when the input argument N is odd:

```
      LOGICAL FUNCTION ODD(N)
C***********************************************************************
C   Function returns .TRUE. if N is odd, returns .FALSE. otherwise
C***********************************************************************
      IMPLICIT NONE
      INTEGER N
      IF(N/2*2 .EQ. N)THEN
          ODD = .FALSE.
      ELSE
          ODD = .TRUE.
      ENDIF
      RETURN
      END
```

Alternatively, one can implement a function named EVEN that returns .TRUE. when N is even (make sure to do it). The following main program (which itself contains an example of the block-IF construct) has been written to test FUNCTION ODD:

```
      PROGRAM ODDNUM
C   Driver routine for FUNCTION ODD
      IMPLICIT NONE
      INTEGER NUM
      LOGICAL ODD
      PRINT*, 'Enter the number to be tested: '
      READ*, NUM
      IF(ODD(NUM))THEN
          PRINT*, 'The number is odd.'
      ELSE
          PRINT*, 'The number is even.'
      ENDIF
      END
```

A sample program output follows:

```
Enter the number to be tested: 4
The number is even.
```

After the number (NUM) has been read in, the block-IF statement is executed. The first step is the evaluation of the logical expression of the block-IF structure. The logical expression here consists of a single function reference, i.e. ODD(NUM). Control then passes into the function and the statements within the function are executed. The dummy argument N is assigned the value of the actual argument NUM. (Remember that there is name independence between program units.) Next, the value of N/2*2 is compared with N. Let us take a close look at this step of the function subprogram. In the arithmetic expression

```
    N/2*2
```

division is carried out first, i.e. the expression is equivalent to

```
    (N/2)*2
```

Since both `N` and the constant `2` are integers, integer arithmetic is used in the evaluation of this expression. In particular, any remainder resulting from the division of two integer values is simply discarded. If `N` is even, the remainder of the division process `N/2` is zero, and `N/2` is exactly equal to half of `N`, and therefore `N/2*2` is equal to `N`. When `N` is odd, however, the remainder of the division is not zero and `N/2` is less than half of `N`. For example, if `N` is `3`, then `N/2` is `1` (which is less than `1.5`) and `N/2*2` has the value `2` which is different from `N`. Thus, if the value of the relational expression

```
N/2*2.EQ.N
```

is `.TRUE.`, then `N` is even. Otherwise, `N` is odd. The name of the function is assigned the value `.TRUE.` if `N` is odd; otherwise it is set equal to `.FALSE.`. Control is then returned to the calling (i.e. main) program. All of this happens during the evaluation of the logical condition of the block-`IF` structure in the main program. If this logical expression evaluates to `.TRUE.`, then the string `'The number is odd.'` is displayed. Otherwise, the string `'The number is even.'` is displayed.

The single-alternative and the double-alternative forms discussed earlier are special cases of the **multiple-alternative decision structure** (also called block-`IF`). The general form of the block-`IF` structure is as follows:

```
IF(logical expression_1)THEN
     Task_1
ELSE IF (logical expression_2)THEN
     Task_2
       .
       .
ELSE IF (logical expression_n)THEN
     Task_n
ELSE
     False Task
ENDIF
```

The *logical expression*$_1$ is first evaluated. If it is true, then *Task*$_1$ is executed and all the other tasks are skipped. If *logical expression*$_1$ is false, then *Task*$_1$ is skipped and *logical expression*$_2$ is next evaluated. Thus, *logical expression*$_1$, *logical expression*$_2$, etc. are evaluated until an expression, say *logical expression*$_k$, that evaluates to true is reached. In that case *Task*$_k$ is executed. If none of the conditions holds true, *False Task* is evaluated. In all cases, execution is continued with the first statement following `ENDIF`. If *False Task* is empty, that is, if there are no statements to be executed when all the conditions (*logical expression*$_1$ through *logical expression*$_n$) evaluate to false, then the word `ELSE` may be omitted.

<u>Example 2.3:</u>

The sign function sgn(x) is commonly defined as follows:

sgn(x) = 1 if x is greater than zero or if x is zero
sgn(x) = -1 if x is strictly less than zero

The implementations of the FORTRAN sign transfer functions ISIGN, DSIGN, and SIGN are based on the above definition of the sign function. In particular, ISIGN(1,0) is equal to 1 and SIGN(1.,0.) is 1., because sgn(0) = 1 according to its definition. Suppose a certain application requires that we take sgn(0) to be zero. In such a case, we cannot directly use the library functions in our computations. We can, however, implement our own sign transfer function. Here is one way this can be done:

```
      PROGRAM MYSIGN
C  Main program to test ISIGN2
      IMPLICIT NONE
      INTEGER NUMBER, ISIGN2
      PRINT*, 'Please enter a number: '
      READ*, NUMBER
      PRINT*, 'Sign =', ISIGN2(1, NUMBER)
      END
C
      INTEGER FUNCTION ISIGN2(M, N)
C*********************************************************************
C  User-defined sign transfer function
C  Works with integer arguments only
C*********************************************************************
      IMPLICIT NONE
      INTEGER M, N
      IF(N.LT.0)THEN
          ISIGN2 = -ABS(M)
      ELSE IF(N.EQ.0)THEN
          ISIGN2 = 0
      ELSE
          ISIGN2 = ABS(M)
      ENDIF
      RETURN
      END
```

Note that this function handles integer arguments only. You must write two other functions, say DSIGN2 and SIGN2, to handle double precision and real arguments, respectively[4].

Exercises:

1. The library function MOD returns the remainder of the division of its first argument by the second. Modify FUNCTION ODD of Example 2.2 to employ MOD to determine if N is odd or not.

2. Insert the following statement just before the END statement in PROGRAM MYSIGN:

```
      PRINT*, 'ISIGN(1,NUMBER) =', ISIGN(1,NUMBER)
```

Recompile and run the program with the following values: 1, 0, and -2. Compare the values returned by ISIGN with those calculated by ISIGN2.

3. Write a program to read in three real numbers, and determine and display the largest of the three numbers (without using library functions).

4. In Example 1.13 we studied a simple function for the calculation of cube root. The following is a more generalized version of that function:

```
      REAL FUNCTION CBRT(X)
C*********************************************************************
C   Calculates cube root of real number X
C*********************************************************************
```

[4] Recall that generic library functions such as SIGN, SQRT, etc. can be used with different types of arguments. It is not possible to define generic external functions in FORTRAN 77. Fortran 90 brings a solution to this problem by allowing programmers to refer to two or more functions using the same generic name (see Chapter 7).

```
IMPLICIT NONE
REAL X
IF(X.LT.0.)THEN
    CBRT = -EXP(LOG(-X)/3.)
ELSE IF(X.GT.0.)THEN
    CBRT = EXP(LOG(X)/3.)
ELSE
    CBRT = 0.
ENDIF
RETURN
END
```

Note that the algebraic equality $-\sqrt[3]{(-x)} = \sqrt[3]{x}$ is utilized to handle negative values of `X`. Write a main program to test this function. What happens if you use the expression `X**(1./3)` to calculate the cube root of a negative number? (Try it.) Notice also how the zero argument situation is handled. (Remember that logarithm of zero is not defined.)

2.4 Compound Logical Expressions

A relational expression is a special type of logical expression. Logical expressions may also consist of logical constants (`.TRUE.` or `.FALSE.`) or logical variables used by themselves. For example, if `FOUND` is a logical variable, then the statement

```
IF(FOUND)THEN
    PRINT*, 'Solution found.'
ELSE
    PRINT*, 'Cannot find solution.'
ENDIF
```

will print the statement `Solution found.` if `FOUND` has the value `.TRUE.`. If, on the other hand, `FOUND` has the value `.FALSE.` when this segment of the program is being executed, the message printed will be `Cannot find solution.`.

Logical expressions more complex than the simpler expressions just mentioned can be constructed using the following FORTRAN **logical operators**:

.AND.   .OR.   .NOT.   .EQV.   .NEQV.

Given any two logical expressions *logexp1* and *logexp2*, we can form the following **compound logical expressions**:

logexp1.AND.*logexp2*

logexp1.OR.*logexp2*

.NOT.*logexp1*

logexp1.EQV.*logexp2*

logexp1.NEQV.*logexp2*

It should be noted that the words AND, OR, NOT, EQV, and NEQV are preceded and followed by periods. The values of these compound expressions depend on the values of *logexp1* and *logexp2* and are as follows (T and F stand for .TRUE. and .FALSE., respectively):

logexp1	*logexp2*	*logexp1*.AND.*logexp2*	*logexp1*.OR.*logexp2*
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

.NOT. is a unary operator that changes the value of any logical expression from .TRUE. to .FALSE. or from .FALSE. to .TRUE.:

logexp1	.NOT.*logexp1*
T	F
F	T

The operators .EQV. and .NEQV. are used to test logical expressions for equivalence and nonequivalence, respectively. For example, *logexp1*.EQV.*logexp2* has the value .TRUE. if *logexp1* and *logexp2* have the same value (both .FALSE. or both .TRUE.). The properties of these operators are summarized below:

logexp1	*logexp2*	*logexp1*.EQV.*logexp2*	*logexp1*.NEQV.*logexp2*
T	T	T	F
T	F	F	T
F	T	F	T
F	F	T	F

Note that the analogous relational operators .EQ. and .NE. are used to compare arithmetic (integer, real, double precision, complex) and character data, whereas .EQV. and .NEQV. are used to compare logical values. The latter two operators are most often used to simplify the structure of logical expressions. The following two expressions, for example, are equivalent in their effect:

```
(A.LE.B.AND.Y.GT.X)  .OR.  (A.GT.B.AND.Y.LE.X)
A.LE.B  .EQV.  Y.GT.X
```

Assume that `A`, `B`, and `C` are real variables, `NAME1` and `NAME2` are of type character and `FLAG` is of type logical. The following are valid logical expressions:

```
A.GT.0.0
(A.LE.B).AND.(C.GT.1.)
FLAG
.NOT.FLAG
(A.LT.2.) .OR. (.NOT.FLAG)
NAME1.NE.NAME2
```

Any combination of the above logical expressions using `.AND.` and `.OR.` are also valid logical expressions. For example

```
((A.LT.2.).OR.(.NOT.FLAG)) .AND. (NAME1.NE.NAME2)
(A.GT.0.0) .AND. (.NOT.FLAG)
```

Similarly, any combination of the above expressions using `.EQV.` and `.NEQV.` are valid logical expressions. For example

```
(NAME1.NE.NAME2) .EQV. (.NOT.FLAG)
(((A.LT.2).OR.(.NOT.FLAG)).AND.(NAME1.NE.NAME2)).NEQV.(A.GT.0)
```

For clarity, extra parentheses and blanks have been used in some of the above expressions. For example, the following three expressions have the same meaning for the compiler:

```
(A.LT.2.) .OR. (.NOT.FLAG)
(A.LT.2.).OR.(.NOT.FLAG)
A.LT.2..OR..NOT.FLAG
```

In forming complicated logical expressions, the following hierarchy of operations (**precedence of operators**) should be remembered:

1. All subexpressions within parentheses are evaluated first. In the case of nested parenthesized subexpressions, the innermost subexpression is evaluated first.
2. A parenthesis-free subexpression is evaluated using the following hierarchy:
   - i) Arithmetic operations
     - a) First precedence: **
     - b) Second precedence: *, /
     - c) Third precedence: +, -
   - ii) Relational operators (.EQ., .NE., .LT., .GT., .GE., .LE.)
   - iii) Logical operators
     - a) First precedence: .NOT.

b) Second precedence: .AND.

c) Third precedence: .OR.

d) Last precedence: .EQV., .NEQV.

3. Operators within the same parenthesis-free subexpression and at the same level of hierarchy (such as .EQ. and .LE.) are evaluated from left to right.

We see that arithmetic operations are performed first. Relational expressions are next evaluated before compound logical expressions. Among the six relational operators, there is no priority and relational operations are carried out from left to right. When in doubt, it is better to use parentheses to explicitly specify the desired grouping of operands and operators. Furthermore, it is recommended that extra parentheses and blank spaces be used when such usage enhances the clarity and legibility of a program. For example, assuming X, Y, Z, A, B, and C are real variables, the expression

```
(A.LT.B)  .OR.  ( (A.EQ.C) .AND. (B.NE.(X+Y/Z)) )
```

and its equivalent

```
A.LT.B .OR. (A.EQ.C .AND. B.NE.(X+Y/Z))
```

are easier to understand than the (also equivalent and correct) statement

```
A.LT.B.OR.A.EQ.C.AND.B.NE.X+Y/Z
```

<u>Example 2.4:</u>

In this example we will look at a function subprogram that can be used to determine if a given year is a leap year or not. Remember that a leap year is a year containing 366 days with February 29 as the extra day. The function is named LEAPYR, and is a logical function that returns .TRUE. if the given year is a leap year; returns .FALSE. otherwise.

A given year is a leap year if it is evenly divisible by 4 and not by 100, or it is evenly divisible by 400. For example, 1999 is not a leap year (1999 is not evenly divisible by 4), whereas 2000 is a leap year (evenly divisible by 400). Similarly, year 1900 is not a leap year (it is evenly divisible by 100 but not by 400). Notice how a compound logical expression is used in the program to implement the definition of a leap year.

```
      LOGICAL FUNCTION LEAPYR(YEAR)
C*****************************************************************************
C  Returns .TRUE. if YEAR is a leap year; returns .FALSE. otherwise
C*****************************************************************************
      IMPLICIT NONE
      INTEGER YEAR, REM4, REM100, REM400
      REM4 = MOD(YEAR, 4)
      REM100 = MOD(YEAR, 100)
      REM400 = MOD(YEAR, 400)
      IF( (REM4.EQ.0 .AND. REM100.NE.0) .OR. REM400.EQ.0)THEN
          LEAPYR = .TRUE.
      ELSE
          LEAPYR = .FALSE.
      ENDIF
      RETURN
      END
```

The function employs the variables named REM4, REM100, and REM400 to store the computed remainders of the year after division by 4, 100, and 400 respectively. The extra parentheses around the expression REM4.EQ.0 .AND. REM100.NE.0 are actually not necessary since .AND. has higher precedence than .OR.. Similarly, extra blanks are used within the compound logical expression to improve readability. The following version, therefore, would also be perfectly valid (although less legible):

```
      IF(REM4.EQ.0.AND.REM100.NE.0.OR.REM400.EQ.0)THEN
```

Note that the following version would also give correct results in this case (why?):

```
      IF(REM4.EQ.0 .AND. (REM100.NE.0 .OR. REM400.EQ.0) )THEN
```

The following is a main program that can be used to test FUNCTION LEAPYR:

```
      PROGRAM LEAP
      IMPLICIT NONE
      INTEGER YEAR
      LOGICAL LEAPYR
      PRINT*, 'Enter the year to be tested: '
      READ*, YEAR
      IF(LEAPYR(YEAR))THEN
          PRINT*, 'It is a leap year.'
      ELSE
          PRINT*, 'No, it is not a leap year.'
      ENDIF
      END
```

Example 2.5:

This example provides another illustration of the use of compound logical expressions constructed using the operators .AND. and .OR..

```
      PROGRAM LETGRD
C   Program determines and prints the letter grade
C   corresponding to a score between 0 and 100.
      IMPLICIT NONE
      INTEGER SCORE
      CHARACTER*1 GRADE, LETTER
      PRINT*, 'Enter the score (between 0 and 100): '
      READ*, SCORE
      LETTER = GRADE(SCORE)
      IF(LETTER.EQ.'*')THEN
          PRINT 1, 'Score entered', SCORE, 'is out of range:'
          PRINT 1, 'You must enter a value between 0 and 100.'
      ELSE
          PRINT 1, 'Score =', SCORE, '=> Letter Grade =', LETTER
      ENDIF
    1 FORMAT(1X, A, :, 1X, I4, 1X, A, :, 1X, A)
      END

      CHARACTER*1 FUNCTION GRADE(SCORE)
C*****************************************************************************
C   Returns letter grades  'A', 'B', 'C', 'D', or 'F' for specified
C   SCORE between 0 and 100. Returns '*' if SCORE is out of range.
C*****************************************************************************
      IMPLICIT NONE
      INTEGER SCORE
      IF(SCORE.LT.0 .OR. SCORE.GT.100)THEN
          GRADE = '*'
      ELSE IF(SCORE.GE.90 .AND. SCORE.LE.100)THEN
          GRADE = 'A'
      ELSE IF(SCORE.GE.80 .AND. SCORE.LE. 89)THEN
          GRADE = 'B'
      ELSE IF(SCORE.GE.70 .AND. SCORE.LE. 79)THEN
          GRADE = 'C'
      ELSE IF(SCORE.GE.60 .AND. SCORE.LE. 69)THEN
          GRADE = 'D'
      ELSE IF(SCORE.GE. 0 .AND. SCORE.LE. 59)THEN
          GRADE = 'F'
      ENDIF
      RETURN
      END
```

It may be noted that, the block-`IF` structure within the function could be replaced by

```
      IF(SCORE.LT.0 .OR. SCORE.GT.100)THEN
          GRADE = '*'
      ELSE IF(SCORE.GE.90)THEN
          GRADE = 'A'
      ELSE IF(SCORE.GE.80)THEN
          GRADE = 'B'
      ELSE IF(SCORE.GE.70)THEN
          GRADE = 'C'
      ELSE IF(SCORE.GE.60)THEN
          GRADE = 'D'
      ELSE
          GRADE = 'F'
      ENDIF
```

This version is simpler (requires less typing). It was decided, however, that explicitly specifying the range of scores that gets each letter grade made the logic of the program more easily understandable.

Example 2.6:

Logical expressions are most frequently used to specify conditions in decision statements, and (as we shall see in the next chapter) in `DO WHILE` loops. They may also be used in assignment statements involving logical variables. This is usually done to simplify the listing of conditions in a block-`IF` structure. An example is provided below.

```
      PROGRAM ORDER
C  Reads in three numbers and determines their relative sizes
C  >= means greater than or equal to
C  >  means strictly greater than
      IMPLICIT NONE
      REAL X, Y, Z
      LOGICAL L1, L2, L3
      PRINT*, 'Enter X, Y, Z: '
      READ*, X, Y, Z
      L1 = X.GE.Y
      L2 = X.GE.Z
      L3 = Y.GE.Z
      IF(L1 .AND. L3)THEN
          PRINT*, 'X >= Y >= Z'
      ELSE IF(L1 .AND. L2)THEN
C         L1.AND.L2 is .TRUE., but L1.AND.L3 is .FALSE.
C         This can happen only if L3 is .FALSE, whereas L1 and L2 are .TRUE.
          PRINT*, 'X >= Z > Y'
      ELSE IF(L1)THEN
C         L1 is .TRUE., but both L1.AND.L2 and L1.AND.L3 are .FALSE.
C         This can happen only if both L2 and L3 are .FALSE
          PRINT*, 'Z > X >= Y'
      ELSE IF(L2)THEN
C         L2 is .TRUE., but L1 is .FALSE.
          PRINT*, 'Y > X >= Z'
      ELSE IF(L3)THEN
C         L3 is .TRUE., L1 and L2 are .FALSE.
          PRINT*, 'Y >= Z > X'
      ELSE
C         L1, L2 and L3 are all .FALSE.
          PRINT*, 'Z > Y > X'
      ENDIF
      END
```

A few sample runs are shown below:

```
Enter X, Y, Z: 1.  2.4  3.0
Z > Y > X

Enter X, Y, Z:  4   2   3
X >= Z > Y

Enter X, Y, Z: -1  2  0
Y >= Z > X
```

By the way, the method used in this program is not a particularly good way of ordering three numbers according to their sizes. It is used here merely to exemplify the application of logical expressions.

2.5 The Logical IF Statement

Frequently, the *Task* in the single-alternative decision structure

```
IF(logical expression)THEN
    Task
ENDIF
```

consists of a single FORTRAN statement. In such a case, the more compact **logical IF statement** can be used. The general form of logical IF is as follows:

IF(*logical expression*) *dependent statement*

The *logical expression* is first evaluated. If it is true, the *dependent statement* is executed. Otherwise, the *dependent statement* is skipped. The *dependent statement* may be any executable statement except another logical IF statement or a control structure (such as a block-IF or a DO loop).

The logical IF statement is the "ancestor" of the block-IF construct, and was actually the most powerful decision-making statement in FORTRAN before the advent of FORTRAN 77[5]. It is still very useful and examples of its application can be found in the subsequent chapters.

2.6 Nested IF Blocks

It is possible to have one block-IF statement *lying completely* within another block-IF statement. Block-IF statements occurring in this manner are said to be **nested.** In forming nested structures, the following rule must be observed: *All nested structures must be wholly contained within a single statement group of the structure(s) they appear in.* With reference to the multiple-alternative decision structure described in Section 2.3, each task (statement group) may contain one or

[5] A new decision making form, namely the CASE construct, has been introduced in Fortran 90.

more block-IF statements, but the inner blocks must not overlap two or more tasks of the outer structure.

Example 2.7:

The following program is a considerably generalized version of PROGRAM QUAD (cf. Example 1.7) which we have studied earlier.

```
      PROGRAM QUAD3
C***********************************************************************
C  Program to solve a quadratic equation of the form
C            A*X**2 + B*X + C = 0
C  The coefficients A, B, and C are real.
C***********************************************************************
      IMPLICIT NONE
      REAL A, B, C, ROOT1, ROOT2, DISCR, REALPT, IMAGPT, EPS
C***********************************************************************
C  We will accept that DISCR=0 if ABS(DISCR).LE.EPS
C***********************************************************************
       EPS = 1.0E-06
C***********************************************************************
C   Read in the coefficients of the equation.
C***********************************************************************
      PRINT*, 'Enter A, B, C: '
      READ*, A, B, C
C***********************************************************************
C   Check if the user-entered coefficients define a quadratic equation.
C   If not, take appropriate action and print informative messages.
C***********************************************************************
      IF(A.EQ.0.)THEN
          IF(B.EQ.0.)THEN
              IF(C.EQ.0.)THEN
                  PRINT*, 'A = B = C = 0 => Trivial equation: 0 = 0'
              ELSE IF(C.NE.0.)THEN
                  PRINT*, 'The coefficients entered are absurd:'
                  PRINT*, 'A = B = 0 but C =', C
                  PRINT*, 'cannot satisfy A*X**2 + B*X + C = 0'
              ENDIF
          ELSE IF(B.NE.0.)THEN
              ROOT1 = -C/B
              PRINT*, 'A = 0  => B*X + C = 0 (a linear equation)'
              PRINT*, 'The solution is -C/B =', ROOT1
          ENDIF
          STOP
      ENDIF
C***********************************************************************
C   Determine the roots of the quadratic equation.
C***********************************************************************
      DISCR = B**2 - 4.0*A*C
      IF(DISCR.LT.-EPS)THEN
          PRINT*, 'There are no real roots.'
          REALPT = -B/(2*A)
          IMAGPT = SQRT(ABS(DISCR))/(2*A)
          PRINT 1, 'First   root =', REALPT, ' + ', IMAGPT, 'i'
          PRINT 1, 'Second  root =', REALPT, ' - ', IMAGPT, 'i'
    1     FORMAT(1X, A, F7.3, A, F7.3, A)
      ELSE IF(ABS(DISCR).LE.EPS)THEN
          PRINT*, 'There is a single real root. '
          ROOT1 = -B/(2*A)
          PRINT*, 'The root is ', ROOT1
      ELSE
          ROOT1 = (-B + SQRT(DISCR))/(2.0*A)
          ROOT2 = (-B - SQRT(DISCR))/(2.0*A)
          PRINT*, 'There are two distinct real roots:'
          PRINT*, 'First  root =', ROOT1
          PRINT*, 'Second root =', ROOT2
      ENDIF
      END
```

Notice how indention is employed to improve the legibility of the program and to make the logic of the block-IF structures clearer. Note also how indention is especially helpful in the case of nested structures. Although it is neither required nor universally obeyed, many professional programmers follow the practice of using two or four blank characters as indention. It is recommended that you use at least two blanks for this purpose.

The program first checks if the coefficients entered by the user of the program define a quadratic equation: The equation is quadratic only if the coefficient of x^2, i.e. a, is nonzero.

If a is zero, then we are not really dealing with a quadratic equation, but the program still examines various possibilities and prints appropriate informative messages. For example, if a is zero but b is nonzero, then we have an equation of the form $bx+c=0$ which can be solved for x.

If both a and b are zero, the equation reduces to the form $c=0$ which is trivially satisfied if the user-specified value of c is indeed zero. This is obviously not a very interesting case, but is nevertheless correctly identified and dealt with by the program.

The remaining possible case when a is zero, i.e. c being nonzero while both a and b are zero, constitutes a *user-error.* The program detects this error and prints out an error message stating that the coefficients entered cannot satisfy the equation $ax^2+bx+c=0$.

It should be remembered that real arithmetic on a computer is only an approximation. Two numbers which are mathematically equal, for example, may differ very slightly if they have been calculated in different ways. As a result, it is usually not sensible to compare two real numbers for equality. The program, therefore, checks if $|b^2-4ac|<\varepsilon$ (where ε is a very small number) rather than testing for the equality $b^2=4ac$ (or $b^2-4ac=0$) to establish that the discriminant is essentially zero and there is a single real root.

One final point that should be made about this program is that it is *not* a particularly good program for calculating the roots of a quadratic equation. It is presented here mainly to demonstrate the use of nested block-IF statements. A better solution of the problem would require a more careful consideration of the limits of *computer arithmetic.* Certain values of a, b, and c, for example, can cause overflow and the program to abort, while certain other values may result in underflow and subsequent incorrect results. In certain other cases, round-off errors caused by the subtraction of two nearly equal values may cause a significant error in one of the calculated roots[6].

Exercise:

5. Write a main program that will read four sides (say, a, b, c, and d) and will determine whether these four sides could form a polygon. To form a polygon, the entered sides should satisfy all of the following conditions:

$0 \le a,\ 0 \le b,\ 0 \le c, 0 \le d$ (at least three of them should be strictly positive)
$a \le b+c+d$
$b \le a+c+d$
$c \le a+b+d$
$d \le a+b+c$

[6] A detailed discussion of the quadratic equation can be found in *Numerical Recipes* by Press et al. and *Fortran 90/95 for Scientists and Engineers* by Chapman.